Testing and Troubleshooting

Goals of this §lab:

- To learn basic strategies for testing and troubleshooting.

Prerequisites: None

©2005-2007 David Byers

Table of Contents

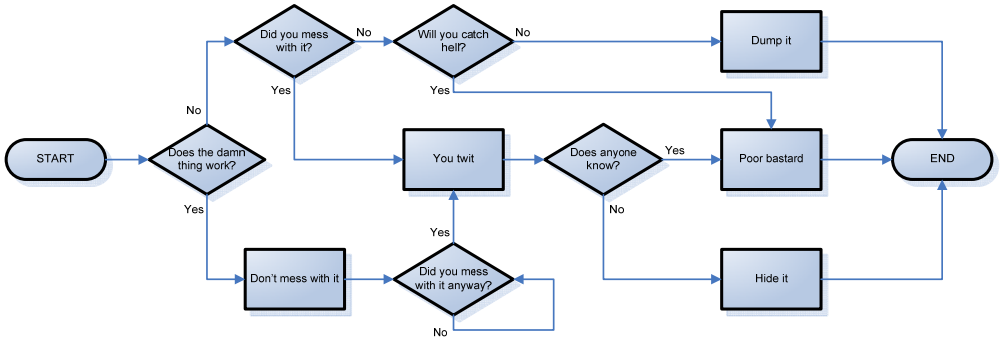
PREPARATION 1

MAIN LAB 3
Part 1: Why so complicated? 3
Part 2: Before troubleshooting 4
Does the damn thing work? 4
Did you mess with it? What did you do? 4
Part 3: Troubleshooting and fixing 5
Reporting the problem 5
Troubleshooting the problem 7
Fixing the problem 8
Part 4: Testing 8
Test cases 8
Test case examples 9
Creating test cases 10
Test protocols 11
Part 5: Change control 11
Proposing the change 11
Reviewing, approving and scheduling changes 14
Following up on changes 14

PREPARATION

This lab has no preparation exercises.

MAIN LAB



In the system administrator's world, testing and troubleshooting are more or less two sides of the same coin. When something goes wrong, finding the problem generally involves testing the system to find out *exactly* what isn't working and what is working.

Getting good at testing and troubleshooting takes practice, and for most people it is more of an art than a science. Nevertheless, there are a number of general strategies that can (and should be applied). These strategies aren't specific to system administration – for the most part they hold true in any kind of testing and troubleshooting.

You will be expected to apply the theory in this lab in all the other labs. In fact, one of the most common reasons for failing a lab in the past has been insufficient testing.

Time taken 2005: 0.5-1.5 hours, average 1 hour

Past problems: None.

Part 1: Why so complicated?

People who have acquired some skills in testing and troubleshooting, but still have limited experience often think treating troubleshooting and testing as formal activities, with processes and documentation and checklists and whatnot is excessive. It's about solving the problem, not documenting it! Why waste time doing all this work when we can just dive in, deal with the issues and get done with it?

The truth is that barging in and just dealing with the issues often works and often works quite well, and that is what makes the ad-hoc approach so attractive. When it works it's fast, cheap and effective. But the ad-hoc approach has several serious shortcomings.

Some shortcomings become obvious when the ad-hoc approach doesn't work. Ad-hoc testing and troubleshooting almost always means a lot of wasted time trying things that never had a chance of working and it frequently means introducing new problems immediately or further down the line, caused by changes that were never very well thought through in the first place.

Even when the ad-hoc approach works, it has serious shortcomings. When the problem is non-trivial, the ad-hoc approach often fails to identify the problem and solution completely. Even when a solution is found, it is rarely evident which parts of the solution were truly necessary and which did not really contribute to solving the problem. This means that the entire problem may not be fixed and the solution may do more than just address the problem, thus becoming a source of future problems.

The ad-hoc approach is short-sighted as it does not prepare for the next time the same or a similar problem appears. The second time a problem appears it will usually be treated as a new problem. The person who solved it the first time may not be on hand or may have forgotten about the original incident, or lack of fully understanding problem and solution may make it difficult to apply the old solution. And so the ad-hoc approach wastes even more time.

The ad-hoc approach doesn't scale well. It doesn't allow troubleshooting to be treated as a team effort, as much of the information needed to perform is locked away in someone's brain. It doesn't support troubleshooting over a long period of time, and it doesn't support a large volume of problems.

More formal processes overcome these shortcomings, but do introduce overhead. In the long run formal processes tend to cost less than ad-hoc approaches, but the savings require up-front investment in time *not* spent "dealing with the issues". Many people have a hard time seeing past the up-front investment to the time savings made later, and many people are overcome by the seductive nature of the ad-hoc approach: even when wasting chasing dead ends, it *feels* like productive work.

Nevertheless, there are situations in which the ad-hoc process is appropriate, but it takes experience to identify them accurately, and it is better to use a formal process once too often than to use an ad-hoc approach on a problem it is unsuited to.

Formal processes don't have to be wasteful. They *shouldn't* be wasteful. The approach I have outlined here is actually fairly lightweight and can result in significant time savings at fairly low actual cost. In real-life situations, where the up-front investment of formal processes has been accepted, it is not uncommon to see processes that have higher overhead, but also result in a higher success rate with fewer problems caused by bad solutions.

Part 2: Before troubleshooting

Does the damn thing work?

That's actually a good question. Sometimes when we *think* something is broken, it isn't. Sometimes our perception of how it should work is at fault. That means that the first question to ask is what the thing is *supposed* to do. Unless you know the answer to that question, it will be very difficult to get any further in the process.

When answering this question we may find that although the thing performs in accordance with its requirements, the requirements are wrong.

> A user is complaining that he can't receive e-mail from a law firm. On examination it turns out that only e-mails containing Microsoft Word documents are not being delivered. Further examination of the documentation for the mail system reveals that the system is *designed* to block Word Documents in order to stop the spread of a particularly nasty virus.
>
> This is an example of where the system is operating as per requirements, but the requirements are not in accordance with the needs of the users. Fixing the problem requires a change to the design of the system. While certainly feasible, it needs to be done carefully, since other parts of the system may depend on the current behavior.

Knowing what part of a system is supposed to do requires preparation. When the system is built or changed, system documentation should be updated to reflect the current requirements of the system. If this is *not* done, and something goes wrong, fixing the problem will take longer because those dealing with the problem first have to figure out what the system is *supposed* to do.

Did you mess with it? What did you do?

If the damn thing is working, don't mess with it. The fact is that every failure is the result of some kind of change (intentional or not). If a system is working and nothing it depends on ever changes, it will never break. It really is that simple.

The problem is that systems depend on so many things, that it is impossible to prevent all changes. But those changes that *can* be prevented should be prevented. At some point in time, even intentional change will be necessary. Environmental factors, requirements and other external factors will eventually change to the point where a change in the system is necessary. When this does take place, it is important to make changes with the utmost care. Very mature organizations use formal change control processes in order to avoid unexpected problems arising from system

changes. A change control process typically involves documenting each proposed change in detail, evaluating its effect on other systems and planning its execution and testing in detail. Proposed changes are evaluated by a change control board, and if approved, changes are scheduled for implementation. Formal change control drastically reduces the rate of change and drastically reduces problems related to changes.

> A global IT services organization moved from ad-hoc, document-as-you-go changes to a strict formal change control process. Each change would be evaluated the morning after it was proposed. Scheduled changes that did not go as expected were immediately rolled back, and re-submitted (after being fixed). Changes that missed their scheduling window were similarly canceled.
>
> Although the system and network administrators initially resisted the new regime, they soon noticed that failures due to messed-up changes had been completely eliminated. During the first two years the process was implemented, the organization experienced zero failures due to intentional system changes.

Change control is currently outside the scope of this course, but you are encouraged to learn about it anyway. Change control can be applied in any engineering-related discipline (and in many others as well).

The least you should do (and in this course you are *required* to do this) is maintain a log book of all changes made to the system. You may want to maintain a separate troubleshooting log, in which you document any problems you encounter, how you figured out what was wrong and what you did about it.

Your logbook is good for a couple of things. Firstly, when something goes wrong, the log book will help you identify what changes were made prior to the failure (and the log book will help refute the "I didn't change anything" claim of others when something breaks). Those are the ones most likely to be responsible for the problem. Secondly, if you ever have to do something over (for example because a disk crashed and you had no backup), the log book will speed things up quite a bit. Thirdly, when someone else needs to figure out something about a system you manage (you might be sick, on vacation or fired), the log book is a lifesaver.

Part 3: Troubleshooting and fixing

Assuming that you've figured out that something is broken, and have a rough idea of *how* it is broken, you can start troubleshooting. I think of troubleshooting as solving a mystery, and I rely heavily on my intuition and experience to quickly find problems. When intuition and experience fail me, I fall back to a more methodical approach. It takes longer, but rarely fails.

Reporting the problem

Troubleshooting starts here. The better the problem report is, the easier it will be to troubleshoot. Problems must be stated with specificity. End users (also known as customers) are often really bad at this. Engineers and scientists should be better, but for some reason, outside their own domains, they're often just as bad as everyone else.

When you're about to troubleshoot things yourself you probably won't write a problem report. After all, you know what the problem is, right? In reality, writing a problem report often clarifies issues, and brings gaps in your understanding of the problem to the forefront. I recommend writing a problem report even if you plan on troubleshooting the issue yourself.

Some of the guidelines I try to adhere to when reporting a problem (or trying to get a decent problem report out of a user):

System details

I always include details of the system (unless I know saying "Linux" will confuse tech support). Type of computer, operating system type and version, network connection details (type, address),

peripherals that are connected, software that is installed and what security features are in place (such as firewalls, antivirus and so forth). I always try to state if the problem has been experienced on just one computer, or on several systems. In a corporate environment, I usually just name the systems.

Problem details

I try to get as many details about the problem as possible. Always include the time the problem occurred, as closely as possible.

State exactly how the problem was triggered, with as much specificity as possible. Exact commands, addresses, mouse movements and so forth should be included. Do not skip anything.

State exactly what the symptoms are. "It won't work" is not good enough. Symptoms include long delays (state how long and how the computer behaves in the mean time), error messages (include *precise* error messages) and anything the computer does. Symptoms also include things you think it did right.

State what you think should have happened, again with specificity. Often an end user and a system administrator will have slightly different expectations of what the system will do, and it is vital to include those expectations in the problem report.

Reproduction procedure

If you can, include instructions on how to reproduce the problem. An idiot should be able to follow them. If you can't (or won't) determine how to reproduce the problem, state this instead.

Troubleshooting performed

If you have performed any troubleshooting, include details of what you have done, and what the results were.

Theories about the cause

Some people, particularly people with a bit of knowledge and an inflated idea of their own skills, will report their theories of what a problem is instead of the problem itself. That is a bad habit, because the theories are wrong more often than not. If you feel a need to include theories in a problem report, clearly label them as such, and don't forget to report the actual problem.

Examples

The following is a *very* bad (but very common) problem report:

> The internet doesn't work. Can you fix it?

There isn't enough detail to even start the troubleshooting process. One might guess that the problem is that booting the computer, then starting the default web browser, then entering a URL does not result in a web page being loaded, but it could just as easily be any number of things, including a user trying to browse the web using Microsoft Excel.

This problem report is slightly better:

> I can't view some web sites in Internet Explorer. I use Windows 7 Home.

This report has some details, but is lacking key information: which browser, which websites, and what exactly does "can't view" mean. Are the fonts too small? Does the web browser crash? Does it take too long? Is there an error message?

The following report a lot better:

> I start Internet Explorer right after booting my computer, and type www.example.com in the address bar, then hit enter. The IE logo starts spinning, but nothing else happens. There isn't even an error message. I was expecting to see a discussion forum dedicated to breeding toads. After about a minute I give up and hit the stop button. I'm using IE 6.0 on Windows XP Home.

> All other websites I've tried (such as www.google.com, www.msn.com and www.ebay.com) work just fine. I tried turning off the Windows firewall, but that didn't help.

Here there is quite a bit of information to go on. The exact symptoms are documented, some troubleshooting steps are included and the very important fact that the problem is an exception, not the rule, is clearly stated.

Contrast the last example to this one:

> The corporate firewall is filtering www.example.com. Please fix it as I need to research toad breeding for the Royston-Vasey project.

This report is bad because it gives no indication of what the problem is, just what the person experiencing the problem *thinks* it is. The theory fits the facts (as far as we know them), but is only one of many possible explanations.

Troubleshooting the problem

The following is a rough outline of the process you can follow when troubleshooting problems, once you have a reasonable description of the problem.

Reproduce the problem

The first thing to do is to reproduce the problem. If the problem can't be reproduced, then finding the cause can be very, very difficult. During this process you should also try to narrow down what the problem really is.

Isolate the problem

Once the problem is reproducible, try to isolate it. This means eliminating as many interactions as possible and establishing the boundaries of the problem. For example, when troubleshooting a network problem, always use IP addresses instead of names, as this (probably) eliminates interaction with DNS.

In the case of an inaccessible web page you might look at whether the problem is reproducible on all browsers (perhaps it is browser related), from all hosts (perhaps it depends on which computer you use), whether it is related to name resolution (by using IP addresses), whether it is a server issue (by connecting using e.g. `socat` or `telnet` and typing in HTTP manually), whether it depends on the operating system, or if it depends on a firewall (try from outside the firewall).

If during the process of isolating the problem you discover an interaction that *can't* be eliminated, then there is a good chance that the cause of the problem is related in some way to that interaction.

Gather symptoms

Next, figure out what the symptoms are. Collect as many and as varied symptoms as possible. The more symptoms you can find, the less likely it is that there will be more than one possible cause of the problem. That helps narrow down the scope of the problem further. In Linux, the system log files stored in /var/log are very useful as most services output diagnostic information to the logs.

Guess what the problem is (and what it isn't)

The next step is to attempt to infer a cause from the symptoms that have been observed. It is also often useful to use the symptoms to exclude possible causes. Try doing both. First, exclude as many possibilities as you can, then move on to the more probable causes, and test them one at a time. This is where experience really plays a part. An experienced troubleshooter is better at inferring and ranking possible causes than an inexperienced troubleshooter, and that translates to faster problem solving.

Proving or disproving the cause

Each cause is examined in turn. For each cause, figure out what symptoms other than those observed the cause should result in. Concentrate on symptoms that are specific to the cause you are examining, trying to avoid more general symptoms. Next, create test cases that will, if the

cause is the real one, show those symptoms. If they do, then that cause becomes more probable, and will be examined in greater detail. If the test cases fail to provoke the expected symptoms, then the cause is discarded, and the observations from the test cases are added to the overall pile of symptoms.

Eventually, often after several iterations of this process, it is possible to determine what the problem is, and proceed to fix it.

Fixing the problem

It is important to recognize that fixing a problem means changing the system, and changes are where problems happen in the first place. Just because the problem is intended to fix one thing doesn't prevent it from breaking others (or, indeed, actually fixing the problem).

Fixes should be narrow in scope

Fix the problem and nothing but the problem. The more things a fix affects, the more can go wrong. The more that goes wrong, the more time you waste.

Fixes require testing

After a fix is applied, not only should tests be run to check that it (seems to have) fixed the problem; tests should be run on anything else that might be affected to ensure that nothing else broke.

Plan for disaster

Things go wrong. It must always be possible to back out a fix (or a partial fix). Sometimes, backing out a fix means restoring old configuration files. Sometimes more complex steps are needed (such as when a software upgrade fails). Always have a plan for backing out the fix if it turns out to be less than successful. The least you can do is copy any files you change so you can restore them later.

Checking for sufficiency

If the fix appears to be successful, check that the original problem really went away. This is yet another round of testing, this time with broad test cases that may involve other services or systems. This step will show if there are additional problems that need to be fixed.

Part 4: Testing

In the section on troubleshooting, frequent mention was made of testing. Testing is a formal activity that is required in nearly all engineering disciplines, and that is very similar in all disciplines. This section will guide you on how to perform testing at a level that is adequate for your lab reports.

In this course you are expected to test everything you do. Your test cases will be evaluated, and if they are not up to scratch, you will have to re-do them. Experience from previous years is clear: groups that took testing seriously finished the labs faster than those that didn't, simply because they tended to have fewer problems.

Test cases

Testing is based on test cases. A test case is a procedure that tests if some property of a system holds. Try to keep test cases focused. There is a temptation to write few test cases, each of which tests a lot of things. The problem with that is that such test cases are useless to guide and assist troubleshooting. Write many small test cases instead (and a few large ones).

There are plenty of good reasons for creating good test cases, but I think one of the most important is that if test cases are well specified and easy to carry out, regression testing (testing to check that a change hasn't had unintended side effects) becomes far easier and cheaper to perform. Without good test cases, regression testing becomes like testing everything all over again from the beginning.

Test cases need to consist of at least the following parts:

Purpose

Specified what the test case is for. It is tempting to create test cases that test a lot of things at once, but try to avoid that. If a test case that tests lots of things at once fails, you usually don't know what part failed, whereas if a test case that only tests a tiny thing fails, you often have a fair idea of what the problem is even before you start troubleshooting.

Unit under test

Specifies what is being tested. The unit might be a software module, a program, a web site, a server, a service on a server, a network, or just about anything else.

Preconditions

What state the unit under test is in before the test starts. Most tests will be sensitive to initial conditions, so it is important to specify them.

Test procedure

The test procedure specifies the exact steps to take to perform the test. Think of it as a program that will be read by a tired, annoyed, unfocused human. Make it as explicit as possible and make it impossible to misinterpret.

For example, if as part of the procedure the IP address of www.example.com needs to be looked up, write "Run 'host www.example.com' and make a note of the IP address". Don't say "Look up the IP address of www.example.com". The latter leaves too much room for interpretation. The tester might choose to use some tool that doesn't always get the address right.

Similarly, when testing DNS one might be tempted to write something like "Look up a few hosts using the nameserver under test and check that the addresses are right". Again, too much room for interpretation (and mixes in expected results with the test procedure). It is better to specify the exact queries.

Expected results/acceptance criteria

This section specifies what results are expected from the test and what conditions need to hold for the test to pass. Again, be explicit. Anything left up to interpretation will be misinterpreted by somebody.

For example, a lazy test case author who wants to check that a DNS server is working might say "The output is expected to be the correct IP address of the host". The whole point of the test is to check that the DNS server is working. To do that, it is necessary to specify how it *should* work, and not leave that up to the tester.

Test case examples

The following are examples of test cases for a DNS server. First the bad (though I've seen worse):

Purpose:	To check that DNS works.
Unit under test:	The DNS server at 130.236.189.1
Preconditions:	The DNS server is to be loaded with our zone.
Procedure:	Look up a few host names in our zone and make a note of their addresses.
Expected results:	The addresses found during testing are correct.
Pass criteria:	All addresses must be correct.

This test case is bad (again, not the worst I've seen) for at least the following reasons: the purpose is too general; the test case does not adequately address the purpose (it only tests part of the purpose); the preconditions are uninformative and cannot be repeated (what does "our zone" mean); the procedure is too general (which host names you look up may affect the outcome); the

expected results don't actually say what results are expected and the pass criteria rely on the unspecified expected results.

Then the good (well, better):

Unit under test:	The DNS server at 130.236.189.1
Purpose:	To check that A record lookups in the zone sysinst.ida.liu.se work from our network and other networks.
Preconditions:	The DNS server is configured with the sysinst.ida.liu.se zone.
	The zone contains A records for www.sysinst.ida.liu.se, ns.sysinst.ida.liu.se and d1-gw.sysinst.ida.liu.se.
Procedure:	Run the following commands on 130.236.189.1, 130.236.189.12 and on any host not connected to 130.236.189.0/24.
	1. host www.sysinst.ida.liu.se.
	2. host ns.sysinst.ida.liu.se.
	3. host d1-gw.sysinst.ida.liu.se.
Expected results:	For command 1: 130.236.189.6
	For command 2: 130.236.189.2
	For command 3: 130.236.189.48
	The same results are expected on each host that the commands are run on.
Pass criteria:	All results are in accordance with expected results on all hosts.

Note the difference in specificity. The good test case takes a lot longer to write, but is just as fast to execute and it is clear when it passes and when it fails. It could be better. For example, it does not specify the version of `host` to use.

Every test case should test only one (or a small number) of things, because then failures tend to indicate what the problem is (they aid troubleshooting). It is generally better to have a lot of small test cases than just one bit one. Best of all is to have both a lot of small test cases, and then some big ones to run once the small ones pass.

Creating test cases

When creating test cases one of the hardest things is to make sure all requirements are covered.

Consider the following requirement:

> Export /export/files to your clients using NFSv3, read-only, with no special access for root.

There are a number of requirements in there.

- The directory /export/files is to be exported (precondition: it must exist).
- Exporting is to be done using NFS version 3 only.
- The directory shall be exported to the clients.
- The directory shall *not* be exported to any other system (implied).
- Files shall be readable but not writeable on the clients.
- Root shall have no more access than any other user.

Each of these requirements leads to a test case (and possibly more than one). The following set of test cases would be appropriate in this example:

- One that verifies on the server that the directory is exported. It could also verify on a client that the directory appears to be exported, without mounting it.
- One that checks that the NFS version in use is version 3.
- One that checks that clients (all of them) can mount the directory read-only.
- One that checks that the clients (all of them) *cannot* mount the directory read-write.
- One that checks that some other system other than the clients on the same network cannot mount the directory.
- One that checks that some other system on some other network than the clients cannot mount the directory.
- One that checks that when the directory is mounted read-only, root does not have any access beyond what any other user would have had.

So when creating test cases, break down the requirements into simpler, concrete, easily testable requirements, then create test cases for those, then implement the requirements, then execute the test cases. By creating test cases *first* you force yourself to think deeper about the problem than if you started with the implementation.

Finally, re-run test cases whenever the requirements they relate to may have changed. For example, if you are implementing a firewall, it would make sense to re-run any networking-related test cases afterwards, to ensure that nothing undesirable has happened.

Test protocols

Each time a test is run, a test protocol should be written (and in this series of labs, you have to hand them in). The test protocol is simply a documentation of the test. In part it mirrors the test case structure: it specifies what the preconditions actually were, what steps were actually carried out (the `script` command is very useful for this) and what results were actually observed. The test protocol also specifies the time the test case carried out and who did the honors.

Test protocols are particularly important for failed tests, as the information they contain can be used for troubleshooting. For successful tests, one might consider just documenting which test was run and that it passed.

Part 5: Change control

I said that change control is outside the scope of this course, but it's something I think every engineer should learn about before entering the workplace. I *strongly* recommend that you use some form of change control in the labs. It will slow you down but also eliminate a lot of problems that would otherwise take time (and be frustrating).

In the context of system administration, change control is typically used to ensure that only appropriate changes (those that are motivated, do not break things, and are economically defensible) are performed, and that any documentation is kept up-to-date. Changes, in this case, are typically adding, removing, altering, or reconfiguring hardware or software.

When change control is applied, any system change must go through a defined process that involves documenting the proposed change, reviewing and approving the proposed change, scheduling the change, performing the change, and evaluating the change.

While formal change control may seem cumbersome (and often it can be), it is possible to implement change control processes that are lightweight, and easy to use. The benefits of an appropriate change control process always outweigh the cost of the process.

Proposing the change

When a change is proposed it is documented in something often called a "change request". A change request will typically include at least the following:

- A unique identifier for the proposed change
- Summary of the proposed change
- Motivation for the proposed change
- Evaluation of the proposed change against key business and technical factors
- Procedure for implementing the change
- Procedure for verifying that the change was successful
- Procedure for removing the change (or any part thereof) in case of failure

A unique identifier is used for traceability. Often, a protocol is created when implementing the change, and that needs to be tied to the change request. The change summary is helpful for those reviewing the change.

Motivation of the proposed change

This section of a change request explains why the change is necessary (or desirable). If a change cannot be motivated, then implementing it is probably not a good use of resources.

Evaluation against key factors

Every change must be evaluated against key factors, whether they are business related or technical. Exactly which factors are important will vary from business to business. Typically factors include impact on other systems (if the change is implemented, how will it affect other systems during and after the implementation of the change), and cost (how much time and/or money will it cost to implement the change). It is important not only to evaluate the effect of a successful change, but also the potential effect of a failed change.

Procedure for implementing

In order to evaluate a proposed change, it is necessary to see what the change involves. The procedure for implementation should be as detailed as possible, clearly identifying every object that is impacted in some way. For example, when disabling recursion on a name server, one would not say:

```
edit the nameserver configuration files
```

but:

```
add 'allow-recursion: none;' to the options section in /etc/bind/named.conf'
```

There are several reasons for this level of detail. One reason is that in order to evaluate a particular change, it is necessary to know exactly what the change entails. Another, and perhaps more important reason is that by detailing a change, any problems, lack of understanding, or potential impact to other systems becomes much clearer. A rule of thumb that I try to apply is that if I can't write a detailed implementation procedure, then I don't understand the change well enough to implement it safely.

Procedure for verifying (verification procedure)

Once a change is implemented it is necessary to test that it has achieved the expected effects (and hasn't impacted anything else negatively). The verification procedure serves this procedure. It is a detailed recipe for verifying that the change was successful. Without a verification procedure it is impossible to know with any certainty that a given change has been completely successful.

Procedure for removing (backout plan)

One of the most important parts of a change is how to remove it, if it proves unsuccessful. This procedure details exactly how to restore a system to the state it was in before the starting to implement the change, and it should be possible to perform at any point in the implementation procedure. Reasons for applying the backout plan is if implementing the change does not go

according to plan (i.e. doesn't work as expected, or takes longer than planned), or if the verification procedure fails.

Other information

The information listed above is a bare minimum of what a change proposal must contain. In addition to that, one would also expect it to include how long the implementation and verification procedures may take before the backout plan is executed, a list of other changes that must be performed before (or after) this one, a list of computer systems and software affected, and so on.

Example of a change proposal

ID:	546
Summary:	Disable recursive DNS lookups from outside the corporate LAN.
Motivation:	Allowing recursive DNS lookups from outside the corporate LAN is a security risk. It exposes us to the risk of cache poisoning and can be used to amplify denial of service attacks.
Evaluation:	If the change fails so the DNS server is disabled or recursive lookups are disabled entirely, then internal systems will have problems accessing the Internet until the backout procedure is complete. We expect a maximum outage of ten minutes in this case. If the change fails so the DNS server stops responding entirely, then external users will not be able to access our systems until the bckout procedure is complete. We expect a maximum outage of ten minutes in this case. The cost of implementing the change is negligible compared to the cost of ignoring the problem.
Implementation:	Connect to host dns.example.com using ssh Change user to rood Copy /etc/bind/named.conf to /etc/bind/named.conf.546 Open /etc/bind/named.conf in emacs Locate the options section (starts with options and a brace, ends with a brace) On any line within the options section, add the following line: allow-recursion { localnets; }; Save the file and exit emacs Run rndc reload Check /var/log/syslog to ensure that the nameserver reloaded correctly.
Verification:	On host one.example.com, execute: dig +recurse www.google.com @dns.example.com +noall +comments Verify that the ra and rd flags are both present. On any host not on the corporate LAN, execute the above query and verify that rd, but not ra, are present.
Backout:	Copy /etc/bind/named.conf.546 to /etc/named.conf
Backout time:	Implementation 15 min, verification 15 min

In this example we have, in addition to the parts mentioned earlier, a backout time, which specifies how long the implementation and verification may take before the backout plan is executed.

Reviewing, approving and scheduling changes

Before a change can be implemented it must be reviewed and approved. This is usually the job of a change control board. The change control board receives all proposed changes and determines which will be carried out, and when. Exactly how often and by whom changes are reviewed depend on the process. It can involve anywhere from one to tens of people.

When using change control it becomes simple to schedule changes in such a way that the have the least negative business impact as possible. For this reason, the change control board should have the mandate to schedule a window of time during which a particular change may be performed. If the change cannot be performed in this window, it must be re-scheduled.

Following up on changes

Once a change has been successfully completed, it should be reviewed again to identify any lessons to be learned from its implementation. Did the various procedures work as expected (and if not, why not)? Was the estimated time correct? Was the scheduling appropriate (and if not, why not)? Questions such as these are important when improving the change control process itself.

FEEDBACK FORM TST

Complete this feedback form **individually** at the end of the lab and hand it to the lab assistant when you finish. Your feedback is essential for improving the labs. Each student should hand in a feedback form. Do not cooperate on completing the form.

You do not need to put your name on the feedback form. Your feedback will be evaluated the same way regardless of whether your name is on it or not. Your name is valuable to us in case you have made and comments in the last section that need clarifications or otherwise warrant a follow-up.

For each section, please rate the following (range 1 to 5 in all cases).

- **Difficulty:** Rate the degree of difficulty (1=too easy, 5=too difficult)
- **Learning:** Rate your learning experience (1=learned nothing, 5=learned a lot).
- **Interest:** Rate your interest level after completing the part (1=no interest, 5=high interest).
- **Time:** How long did the part take to complete (in minutes)?

	Difficulty	Learning	Interest	Time (minutes)
Part 1: Why so complicated?				
Part 2: Before troubleshooting				
Part 3: Troubleshooting and fixing				
Part 4:Testing				
Part 5: Change control				
Overall				

Please answer the following questions:

- What did you like about this lab?

- What did you dislike about this lab?

- Make a suggestion to improve this lab.

FEEDBACK FORM TST

Complete this feedback form **individually** at the end of the lab and hand it to the lab assistant when you finish. Your feedback is essential for improving the labs. Each student should hand in a feedback form. Do not cooperate on completing the form.

You do not need to put your name on the feedback form. Your feedback will be evaluated the same way regardless of whether your name is on it or not. Your name is valuable to us in case you have made and comments in the last section that need clarifications or otherwise warrant a follow-up.

For each section, please rate the following (range 1 to 5 in all cases).

- **Difficulty:** Rate the degree of difficulty (1=too easy, 5=too difficult)
- **Learning:** Rate your learning experience (1=learned nothing, 5=learned a lot).
- **Interest:** Rate your interest level after completing the part (1=no interest, 5=high interest).
- **Time:** How long did the part take to complete (in minutes)?

	Difficulty	Learning	Interest	Time (minutes)
Part 1: Why so complicated?				
Part 2: Before troubleshooting				
Part 3: Troubleshooting and fixing				
Part 4:Testing				
Part 5: Change control				
Overall				

Please answer the following questions:

- What did you like about this lab?

- What did you dislike about this lab?

- Make a suggestion to improve this lab.